LAPACK in SILC: Use of a Flexible Application Framework for Matrix Computation Libraries

Tamito KAJIYAMA
JST CREST and the University of Tokyo
kajiyama@is.s.u-tokyo.ac.jp

Akira NUKADA
JST CREST and the University of Tokyo
nukada@is.s.u-tokyo.ac.jp

Hidehiko HASEGAWA
University of Tsukuba and JST CREST
hasegawa@slis.tsukuba.ac.jp

Reiji SUDA
The University of Tokyo and JST CREST
reiji@is.s.u-tokyo.ac.jp

Akira NISHIDA
The University of Tokyo and JST CREST
nishida@is.s.u-tokyo.ac.jp

Abstract

This paper presents a novel application framework named Simple Interface for Library Collections (SILC) that allows users to make use of matrix computation libraries in a flexible and language-independent manner. Using SILC, various computing environments as well as alternative solvers and matrix storage formats from different libraries can be easily utilized. The present paper describes the design and implementation of SILC for shared-memory parallel computing environments, and discusses the use of LAPACK in the framework of SILC together with some experimental results on the performance of the implemented system.

1. Introduction

Driven by the vital needs for matrix computations in many scientific and industrial applications, a large number of matrix computation libraries have been developed [2, 6, 12]. User programs that make use of these matrix computation libraries usually prepare data such as matrices and vectors using library-specific data structures, and make a function call by specifying the name of a library function and its arguments in a prescribed order. Although function calls are intuitive and useful in general, utilization of a library by means of function calls necessarily makes the user programs dependent upon the library. This implies that considerable modification in the user programs is necessary to replace the library by another.

Replacement of matrix computation libraries is required (1) when porting user programs to different computing environments, and (2) when using different solvers and matrix storage formats. Large-scale scientific and industrial applications are in need of high-performance computing environments, each of which has its own matrix computation libraries in order to achieve near-peak performance in the computing environment. These special libraries are not available in other computing environments, so that the users who port their user programs to different computing environments are required to switch matrix computation libraries by modifying the source codes of the user programs. Moreover, the most efficient solvers and matrix storage formats vary according to the problems to be solved and the computing environments in use [1]. Since a matrix computation library provides a limited number of solvers and matrix storage formats, it is desirable (and even mandatory in some cases) to try out various matrix computation libraries to find the best solver and matrix storage format for a given problem in a specific computing environment.

Use of matrix computation libraries by means of function calls is a good practice when using a small number of libraries in a particular computing environment. However, when two or more computing environments are used together with a number of alternative solvers and matrix storage formats from different libraries, a source-level dependency upon a specific library imposes heavy burdens on users when they switch libraries.

To address this issue, we have been proposing a novel application framework named Simple Interface for Library Collections (SILC) [3]. Our framework enables user programs to be independent of matrix computation libraries

In Proceedings of the 8th International Conference on High-Performance Computing in Asia-Pacific Region (HPC Asia 2005), pp. 205-212, Beijing, China, November 30-December 3, 2005.

through the following three design decisions: (1) to separate data transfer and request for computation, (2) to request the computation by means of mathematical expressions in the form of text, and (3) to carry out the requested computation in separate memory space independently of user programs. The user programs in the framework of SILC are free of library-specific function calls, so that users can easily switch matrix computation libraries without making a number of modifications to the user programs.

Among various approaches for higher usability of matrix computations, our proposal is mainly related to prior research based on Remote Procedure Call (RPC) and code generation techniques. NetSolve [8] and Ninf–G [9] are middleware that enables RPC to be carried out in Grid environments. User programs for these systems are written in a manner similar to the traditional function calls, while user programs in SILC make use of mathematical expressions to request matrix computations. Telescoping Languages [5] and CMC [4] are source-to-source code translation systems, in which user programs are written in the MATLAB language [11]. The focus of Telescoping Languages is on the optimization of both user programs and matrix computation libraries, whereas CMC focuses on the generation of optimized Fortran subroutines from user programs as well as on rich support for various sparse matrix storage formats. In SILC, on the other hand, the focus is on the use of various matrix computation libraries in a flexible and language-independent manner.

In this paper, we describe the design and implementation of SILC for shared-memory parallel computing environments. We also discuss the use of LAPACK [7] in the framework of SILC as a practical example, with which we clarify the benefits of using matrix computation libraries in the proposed framework.

2. Overview of SILC

We have been developing a SILC system based on a client-server architecture. Figure 1 shows an architectural overview of the implemented system. Assumptions in the current implementation are as follows: (1) a user program (i.e. a client of a SILC server) is a sequential program; (2) the SILC server runs in a shared-memory parallel computing environment; and (3) the libraries to be installed into the SILC server are parallel libraries based on OpenMP.

2.1. User Program (Client)

A user program is invoked together with a SILC server that is started in advance in a local or remote computing environment. The user program establishes a network connection to the SILC server and utilizes the features of matrix

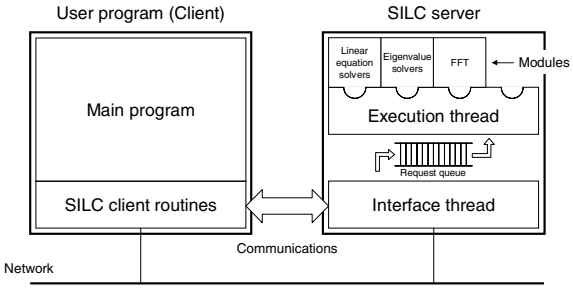


Figure 1. Architectural overview of SILC.

computation libraries (installed into the server as modules) by sending the following three types of requests.

1. At first, PUT requests are used to deposit data such as matrices and vectors. The data is labeled by a user-defined name for later reference and sent to the SILC server through the network connection. The data is kept undeleted in the server's memory space unless deletion is explicitly requested.
2. After the deposit of all relevant data, EXEC requests are used to instruct computation by means of mathematical expressions in the form of text. The names defined by preceding PUT requests are used in the expressions to refer to the deposited data, and each operator that appears in the expressions is translated into a call of a library function, which is carried out in the memory space of the SILC server. The data resulting from the function call is retained in the server's memory space. No memory space in the user program is used for the execution of the library function.
3. Finally, when GET requests are issued, the data in the server side is sent back to the memory space of the user program. Names are used to specify the data to be fetched. The data in the server's memory space remains unchanged even after the GET requests.

Traditional programming languages such as C and Fortran, as well as scripting languages such as Python, can be used for the development of user programs in SILC. In the case of user programs written in C, the following three SILC client routines are used to issue PUT, EXEC and GET requests respectively:

- `SILC_PUT(`⟨*name*⟩`,` ⟨*data*⟩`)`
- `SILC_EXEC(`⟨*expr*⟩`)`
- `SILC_GET(`⟨*data*⟩`,` ⟨*name*⟩`)`

where ⟨*name*⟩ is a name of data and ⟨*expr*⟩ is a mathematical expression (whose syntax is described in Section 2.3), both specified by string. ⟨*data*⟩ is a pointer to the `silc_envelope_t` structure that is used for data communications between the user program and the SILC server.

```
SSI_MATRIX A;
SSI_SCALAR *b, *x, work[N*4], params[2];
int options[6], status;

/* Create a matrix A and a vector b */

status = ssi_cg(b, x, work,
                params, options, &A, NULL);
```

(a)

```
silc_envelope_t A, b, x;

/* Create a matrix A and a vector b */

SILC_PUT("A", &A);
SILC_PUT("b", &b);
SILC_EXEC("x = A \\ b"); /* Solve Ax = b */
SILC_GET(&x, "x");
```

(b)

Figure 2. Comparison of two methods for utilizing matrix computation libraries. (a) is part of a user program written in C that makes use of a library function based on a traditional function call. (b) is another user program in C that carries out the same computation in the framework of SILC.

Figure 2 (a) shows part of a user program that makes use of a matrix computation library based on a traditional function call. Function `ssi_cg` [6] is a library function that solves a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$ using the CG method [1]. The input data of the library function are a matrix A and vector $\boldsymbol{b}$, and the output is a solution $\boldsymbol{x}$. To call the library function, the matrix A and vector $\boldsymbol{b}$ need to be prepared in some data structures that are specific to the library. In addition, the function call is made using a library-specific function name together with a number of arguments in a prescribed order.

On the other hand, Figure 2 (b) shows another user program that carries out the same matrix computation in the framework of SILC. In this framework, the matrix A and vector $\boldsymbol{b}$ are deposited to a SILC server by two separate calls of the client routine `SILC_PUT`, together with a user-defined name for each data (i.e. "`A`" and "`b`" respectively). Then, solution of the system of linear equations is requested by a call of `SILC_EXEC`, whose argument is a mathematical expression in the form of text that instructs the computation.[1] In the expression, the two user-defined names are used to refer to the deposited data, and a new variable name "`x`" is defined to store the solution. The SILC server translates the mathematical expression into a call of an appropriate library function (e.g., `ssi_cg`). Finally, the solution $\boldsymbol{x}$ is fetched by `SILC_GET` with the user-defined name "`x`". The primary difference in contrast to the traditional programming style with function calls is that the user program in SILC does not contain any code that is specific to the actual library to be used for the computation.

2.2. SILC Server

A SILC server is a multithreading program that consists of two threads. One is the interface thread that handles PUT and GET requests from a user program, dealing with all data communications between the user program and the SILC server. The other is the execution thread that handles EXEC requests. The EXEC requests are first received by the interface thread and stored into a queue between the two threads. Then the queued requests are processed one after another by the execution thread. The user program and the interface thread run synchronously, while the execution thread handles the EXEC requests asynchronously. These threads are implemented by the standard pthreads library.

With a SILC server running in a parallel computing environment, user programs can automatically gain benefits of asynchronous parallel computation, even though they are sequential programs. Support for parallel user programs is in our future plans; we will discuss it in Section 5.

In addition, a SILC server can be a "hub" through which separate user programs exchange data, because the data in the server remains undeleted. Therefore, users can easily integrate a series of cooperative user programs in such a manner that PUT requests are issued by a mesh generation program, EXEC requests by a computation program, and GET requests by a visualization program.

2.3. Command Language

User programs in the framework of SILC make a request of computation by means of mathematical expressions in a simple command language. The language is designed with the aim of making it possible to instruct matrix computations in a portable and language-independent manner.

The unit of computation to be carried out at once is a statement, which is either an assignment or a procedure call. The left-hand side of an assignment statement is a variable name, which can be used without declaring a data type. The right-hand side of the assignment is an expression that is composed of variable names, operators and function calls. The operators include binary arithmetic operators (`+`, `-`, `*`, `/`, `%`), solution of a system of linear equations (for example, `A \ b` obtains a solution $\boldsymbol{x}$ as in $A\boldsymbol{x} = \boldsymbol{b}$), elementwise multiplication (`*@`) and division (`/@`) of matrices and vectors, conjugate transposition (`A'`), complex conjugate (`A~`) and

[1] The operator for solution of a system of linear equations is represented by a backslash, which is the escape character in string literals in C. Therefore, the operator is written as "\\" in Figure 2 (b).

subscript (e.g., `A[1:5,1:5]` represents a 5×5 submatrix of `A`). There is no language feature for execution flow control; conditional branching and loops are supposed to be realized by the programming languages in which user programs are written.

The command language makes a distinction between procedures and functions in order to exclude ambiguities from mathematical expressions. Procedures can change the values of arguments, while functions cannot; in other words, the arguments of functions are read-only, whereas the arguments of procedures are considered to be read-write variables. Suppose that in the following statement, `X` is a matrix, and function `f` changes some elements of `X`:

```
A = f(X) + g(X)
```

Then, semantic ambiguity arises with regard to `X` to be passed as the argument of function `g`; the matrix may be passed to `g` before it is modified in `f`, as well as passed after modified in `f`. Therefore, functions are not allowed to change the values of arguments.

Also in order to avoid multiple interpretations of a computation request, an assignment in the command language is a statement instead of an expression, which implies that the following example yields a syntax error:

```
A = f(X) + (X = Y)
```

If this were a valid example, two interpretations would be possible on the value of variable `X` to be passed as the argument of function `f`; namely, `X` may be of the old value before the assignment `X = Y` is carried out, as well as the new value that equals to `Y`. To exclude this kind of ambiguity, an assignment is considered to be a statement in the command language.

As a means of requesting matrix computations, use of mathematical expressions in the form of text has the following advantages.

- Mathematical expressions are simple. They are often easier to read and more intuitive than the traditional function calls in C and Fortran.
- Mathematical expressions are well-defined and very portable. They are uniformly interpreted regardless of matrix storage formats, precisions and computing environments.
- Mathematical expressions can be considered a sort of application programming interface (API) that is independent of programming languages. Some matrix computation libraries provide multiple APIs for several programming languages, but the details of the APIs may vary according to the programming languages. Using mathematical expressions for requesting matrix computations, on the other hand, users do not have to care about the differences of APIs among programming languages.
- Text is a simple form of data, so that mathematical expressions in the form of text can be supplied in many ways: as string literals in program codes, by data files, as command-line arguments of user programs, through interactive sessions with users, and so on.

3. Use of LAPACK in SILC

In this section, we take LAPACK [7] as an example of a matrix computation library and discuss the way of using the library in the framework of SILC.

LAPACK (an acronym for Linear Algebra Package) is a Fortran library including a number of solvers for systems of linear equations, linear least square problems, and eigenvalue problems. LAPACK provides support for three matrix storage formats: full, packed and banded. Full storage format is a two-dimensional array that stores all elements of a matrix column by column. Packed storage format is used to store symmetric, Hermitian or triangular matrices compactly, in which only the elements of either upper or lower triangle are stored, being packed by columns. Finally, banded storage format is used for band matrices. A $m \times n$ band matrix with kl non-zero sub-diagonals and ku non-zero super-diagonals are stored in a two-dimensional array with $(kl+ku+1)$ rows and n columns; each row of the array stores a non-zero diagonal, and elements in a column of the matrix are stored in the corresponding column of the array.

There are three modes in which LAPACK is utilized in the framework of SILC as described below.

(A) Both a user program and a SILC server use the matrix storage formats and solvers provided by LAPACK. For example, the user program deposits matrices in the banded storage format, and the SILC server solves systems of linear equations using a pair of LAPACK routines `dgbtrf` (for forming a triangular factorization of a band matrix in double precision) and `dgbtrs` (for solving systems of linear equations with the factored matrix and multiple right-hand sides).

(B) A user program deposits matrices using a storage format of LAPACK, while a SILC server uses solvers provided by other matrix computation libraries. For example, the SILC server can convert matrices from the banded storage format into Compressed Row Storage (CRS) format [1] and use `ssi_cg` (from a library of iterative solvers [6]) to solve systems of linear equations with the CG method.

(C) A user program deposits matrices using a storage format other than the three storage formats of LAPACK, while a SILC server use solvers of LAPACK after converting deposited matrices into one of the three storage formats. For example, the user program prepares a matrix in the CRS format and deposits it to the SILC

```
double *A, *b;
int N, kl, ku, lda, ldb, nrhs, info, *ipiv;

/* Create band matrix A (with kl sub-diagonals and ku super-
   diagonals) and nrhs right-hand sides b */

/* Form a triangular factorization of matrix A */
dgbtrf(N, N, kl, ku, A, lda, ipiv, &info);
if (info == 0) {
  /* Solve systems of linear equations Ax = b with the
     factored matrix, replacing b with solutions */
  dgbtrs('N', N, kl, ku, nrhs, A, lda, ipiv, b,
         ldb, &info);
}
```

Figure 3. A user program written in C with direct calls of LAPACK routines.

server, which converts the matrix into the banded storage format and solves systems of linear equations by `dgbtrf` and `dgbtrs`.

Figure 3 illustrates part of a user program written in C that directly calls two LAPACK routines `dgbtrf` and `dgbtrs` to solve systems of linear equations $A\boldsymbol{x} = \boldsymbol{b}$, where `A` is an $N \times N$ band matrix with kl sub-diagonals and ku super-diagonals in the banded storage format, and `b` is an $N \times nrhs$ full matrix that represents multiple right-hand sides (*nrhs* stands for the number of right-hand sides). After the call of `dgbtrf`, the band matrix in `A` is replaced with a triangular factorization of the matrix. Similarly, `dgbtrs` replaces the right-hand sides in `b` with solutions. Therefore, the copies of the matrix and right-hand sides have to be made if they are used after the calls of the LAPACK routines (for example, in order to check the solutions using the input data).

Figure 4 shows a user program in C that makes use of LAPACK in SILC in the mode (A) above (this example also illustrates the usage of the `silc_envelope_t` structure in more detail than Figure 2 (b)). The band matrix `A` and right-hand sides `b` are created in the same way as in the user program of LAPACK. After the matrix and right-hand sides are deposited to a SILC server by `SILC_PUT`, the solution of systems of linear equations is requested by `SILC_EXEC`. The same LAPACK routines `dgbtrf` and `dgbtrs` are used for the solution, since the two operands of the \ operator are band and full matrices of double precision. Copies of `A` and `b` are automatically made in the SILC server; the user program does not have to care about memory management during the computation.

The following is a summary of major benefits of using LAPACK in the framework of SILC.

- Various solvers and matrix storage formats (possibly of other libraries) can be used without any modification to user programs. Since matrices are sent to the memory

```
silc_envelope_t object;
double *A, *b, *x;
int N, kl, ku, nrhs;

/* Create band matrix A (with kl sub-diagonals and ku super-
   diagonals) and nrhs right-hand sides b */

/* Deposit band matrix A */
object.v = A;
object.type = SILC_MATRIX_TYPE;
object.format = SILC_FORMAT_BAND;
object.precision = SILC_DOUBLE;
object.m = object.n = N;
object.l = kl;
object.u = ku;
SILC_PUT("A", &object);

/* Deposit right-hand sides b */
object.v = b;
object.type = SILC_MATRIX_TYPE;
object.format = SILC_FORMAT_DENSE;
object.precision = SILC_DOUBLE;
object.m = N;
object.n = nrhs;
SILC_PUT("b", &object);

/* Solve systems of linear equations Ax = b */
SILC_EXEC("x = A \\ b");

/* Fetch solutions x */
object.v = x;
SILC_GET(&object, "x");
```

Figure 4. A user program in C that makes use of LAPACK in the framework of SILC.

space of a SILC server, the server has a free choice of matrix storage formats and solvers to be used for computation. The use of different storage formats and solvers in the server does not affect the user programs except for an increase in execution times.

- The same computation request "`x = A\b`" can be used for all the three storage formats as well as for all precisions. LAPACK supports single and double precisions of both real and complex numbers, and the names of LAPACK routines vary according to the storage formats and precisions the routines accept. For example, `dgbtrf` and `dgbtrs` accept real band matrices of double precision as the prefix `dgb` indicates (it stands for "double general band"). In SILC, on the other hand, appropriate solvers are selected according to the storage formats and precisions.
- User programs will be independent of vender-specific C interfaces. Since LAPACK is a Fortran library that is not compatible with the "passing by value" fashion of function calls in C, most vendors of optimized LAPACK libraries (including Intel Math Kernel Library and Sun Performance Library) provide convenient C

Table 1. Computing environments used for experiments.

Environment	Specifications	OpenMP
1. A notebook PC	Intel Pentium M 733 1.1GHz, 768MB memory, Fedora Core 3	N/A
2. IBM eServer xSeries 335	Dual Intel Xeon 2.8GHz, 1GB memory, Red Hat Linux 8.0	2 threads
3. IBM eServer OpenPower 710	IBM Power5 1.65GHz × 2 (4 logical CPUs), 1GB memory, SuSE Linux Enterprise Server 9	4 threads
4. SGI Altix3700	Intel Itanium2 1.3GHz × 32, 32GB memory, Red Hat Linux Advanced Server 2.1	16 threads

interfaces to the libraries for the sake of user programs written in C. However, there is no de facto standard among the C interfaces, so that the user programs in C will depend on a certain vendor-specific C interface; it is not the case in SILC.

4. Experiments

This section shows the results of two experiments on (1) the performance of SILC systems in different computing environments and (2) the performance of a user program that utilizes LAPACK in the framework of SILC. Table 1 is a summary of four computing environments used in the experiments. The notebook PC of Environment 1 and the other three of Environments 2 through 4 are interconnected via a 100 Base–TX local-area network.

4.1. Performance of SILC in different computing environments

We examined the performance of SILC's client-server architecture by using the two user programs shown in Figure 2 (a) and (b). We call them Programs (a) and (b) for short. In this experiment, these user programs are used to solve a five-point discrete Laplacian on $n \times n$ uniform orthogonal grid with zero Dirichlet boundary conditions. The matrix A of the corresponding system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$ to be solved is as follows:

$$A = \begin{bmatrix} T & -I & & \\ -I & T & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & T \end{bmatrix}_{N\times N}, \quad T = \begin{bmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{bmatrix}_{n\times n}$$

where $N = n^2$ and I is the $n \times n$ identity matrix. The CRS format is used as the storage format of A, whose number of non-zero elements is $5N - 4n$.

Figure 5 shows the experimental results, with the dimension N of the matrix A on the horizontal axis and the total execution time in seconds on the vertical axis. Both Programs (a) and (b) are executed in Environment 1 shown in Table 1. Results with the label E1 are of Program (a), which directly calls a sequential version of the library function `ssi_cg`. Results with the labels E2 through E4 are of Program (b) together with different SILC servers in Environments 2 through 4, respectively. A parallel version of `ssi_cg` is used in the SILC servers; the number of OpenMP threads in each environment is also shown in Table 1.

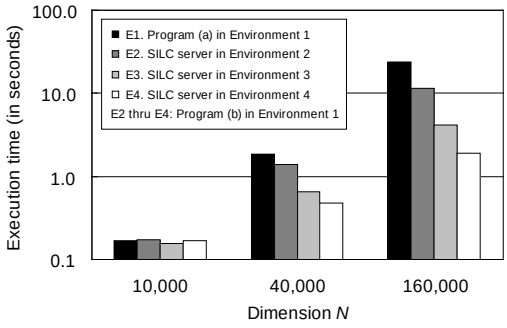


Figure 5. Performance of Programs (a) and (b). E1 is of Program (a) in Environment 1, while E2 through E4 are of Program (b) in Environment 1 together with different SILC servers in Environments 2 through 4, respectively.

In the case of $N = 10000$, the execution time of E2 is slightly longer than that of E1, because two calls of `SILC_PUT` for depositing A and $\boldsymbol{b}$ and a call of `SILC_GET` for fetching $\boldsymbol{x}$ impose some overhead (about 0.1 second) due to data communications, which happened to be greater than the performance gain resulting from parallel computation with 2 threads in the SILC server in Environment 2. In all of the other cases, Program (b) achieved better performance than Program (a), because the communication overhead in SILC is fully canceled by the speed-ups through parallel computation with multiple OpenMP threads.

4.2. Performance of LAPACK in SILC

We also examined the performance of user programs that make use of LAPACK in the framework of SILC based on the three modes (A) through (C) described in Section 3. Since LAPACK is a sequential matrix computation library for dense matrices, use of LAPACK in the modes (A) and (B) is unfavorable to SILC because of relatively large overhead resulting from data communications. As an example, suppose that a user program deposits the matrix A in the previous example in the banded storage format and requests a SILC server to solve $A\boldsymbol{x} = \boldsymbol{b}$ using two LAPACK routines `dgbtrf` and `dgbtrs`. The matrix A in question consists of

five non-zero diagonals, having a lot of zero elements between outer non-zero diagonals, so that depositing the matrix in the banded storage format will impose heavy communication overhead when the dimension N is large. For instance, in the case of $N = 10000$, the matrix A of double precision in the banded storage format takes 23MB, while the data size of the same matrix in the CRS format is only 620KB. Moreover, LAPACK is a sequential library, so that the overhead of data communications cannot be canceled with parallel computation in the SILC server. Therefore, it is not a good idea in general to use LAPACK in the framework of SILC in the modes (A) and (B).[2]

On the other hand, use of LAPACK in the mode (C) has a potential for good performance. To substantiate it, we conducted another experiment using the following two user programs, together with the same matrix A used in the experiment in Section 4.1:

Program #1 is a user program in the framework of SILC that deposits the matrix A of dimension 40,000 in the CRS format and issues the following computation request:

```
X = band(A) \ B
```

where `band` is a function that converts the matrix passed as the argument into LAPACK's banded storage format, and `B` is a full matrix that stores multiple right-hand sides $\boldsymbol{b}_1$, $\boldsymbol{b}_1$, ... $\boldsymbol{b}_{nrhs}$. In the corresponding SILC server, the two LAPACK routines `dgbtrf` and `dgbtrs` are used to solve $A\boldsymbol{x} = \boldsymbol{b}_1$, $A\boldsymbol{x} = \boldsymbol{b}_2$, ..., $A\boldsymbol{x} = \boldsymbol{b}_{nrhs}$.

Program #2 is a user program that makes multiple calls of the library function `ssi_cg` to solve the same systems of linear equations with the CG method. The matrix A is prepared in the CRS format.

Figure 6 shows the experimental results, with the number of right-hand sides (*nrhs*) on the horizontal axis and the total execution time in seconds on the vertical axis. Both Programs #1 and #2 are executed in Environment 1 shown in Table 1, while the SILC server used by Program #1 is in Environment 2. The execution time of Program #2 is proportional to the number of right-hand sides, since each system of linear equations is independently solved. With the two LAPACK routines, on the other hand, solution of *nrhs* systems of linear equations is performed by one call of `dgbtrf` for forming a triangular factorization of the matrix A and one call of `dgbtrs` for solving the systems of linear equations with the factored matrix. Therefore, the performance gain of Program #1 is significant in the cases of $nrhs \geq 4$.

[2]This discussion does not imply that the modes (A) and (B) are useless; LAPACK is merely one of matrix computation libraries that would be used in the framework of SILC, and there are the cases in which use of a library in the modes (A) and (B) achieves good performance.

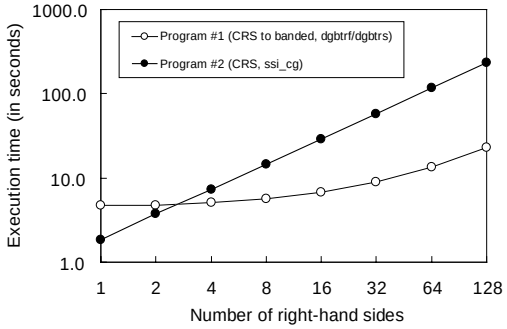


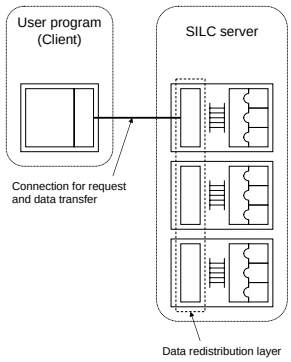
Figure 6. Performance of a user program that utilizes LAPACK in the framework of SILC in the mode (C), compared with a user program that directly makes multiple calls of an iterative solver.

The main benefit proved in this experiment is that in the framework of SILC, it is relatively easy to switch matrix storage formats and solvers to be involved in the solution of linear systems. Users only needs to change mathematical expressions issued by `SILC_EXEC`. In contrast, in the traditional programming style based on direct function calls, the switch of matrix storage formats often requires considerable modification of user programs, which makes it impractical, for example, to find the most efficient matrix storage format from a number of alternatives, possibly provided by different matrix computation libraries.

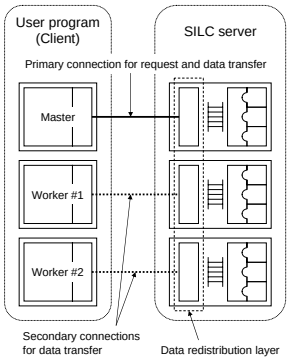
5. MPI-based SILC for distributed-memory parallel computing environments

We have plans for a SILC system for distributed-memory parallel computing environments based on Message Passing Interface (MPI). Figure 7 shows two configurations of MPI-based SILC systems. In the configuration (a), a SILC server is an MPI-based program, while a user program (client) is a sequential program, which would be identical with the user programs in SILC for shared-memory parallel computing environments discussed in the preceding sections. In the configuration (b), on the other hand, both a SILC server and a client are MPI-based programs.

Each of the two configurations needs a mechanism of data redistribution that rearranges data before transferring it between the SILC server and the user program. In the configuration (a), the user program communicates data with one of the server processes. The data to be transferred between the user program and the server process needs to be redistributed among all the server processes so that the server can perform matrix computations on the data in parallel. Similarly in the configuration (b), each of the client



(a) The SILC server is an MPI program, while the user program (client) is sequential.



(b) Both the SILC server and client are MPI programs.

Figure 7. Two client-server configurations of MPI-based SILC systems.

processes sends part of data to one of the server processes. The user program and the SILC server may employ different data distributions, so that if it is the case, the redistribution mechanism is exploited to rearrange the data from one data distribution to the other.

The design and implementation of MPI-based SILC systems together with the data redistribution mechanisms are in our future work.

6. Concluding Remarks

Use of matrix computation libraries in the traditional programming style based on direct function calls usually results in a source-level dependency upon the libraries, making it difficult to switch libraries without modifying user programs. To address this issue, we have proposed a new application framework for using matrix computation libraries in a flexible and language-independent manner. In this paper, we described the design and implementation of the proposed framework for shared-memory parallel computing environments. We also discussed the use of LAPACK in the framework of SILC. Although the discussion in Section 3 was specific to LAPACK, similar discussions apply to most matrix computation libraries other than LAPACK. In brief, SILC makes user programs independent of matrix computation libraries to be used, allowing users to easily switch libraries without making a considerable amount of modification to the user programs.

As discussed in the previous section, our next step is to develop an MPI-based SILC system for distributed-memory parallel computing environments. We also plan to provide modules for major matrix computation libraries, allowing users to easily switch libraries and compare the performances of user programs with respect to different solvers and matrix storage formats. Implementation of a scripting language for SILC and run-time optimization of mathematical expressions are also in our future plans.

Acknowledgments

This research [10] was supported by a grant from the Core Research for Evolutional Science and Technology (CREST) of Japan Science and Technology Agency (JST).

References

[1] R. Barrett *et al*. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[2] J. Dongarra. Freely available software for linear algebra on the Web, May 2004. http://www.netlib.org/utk/people/JackDongarra/la-sw.html.

[3] T. Kajiyama, A. Nukada, H. Hasegawa, R. Suda, and A. Nishida. SILC: a flexible and environment independent interface to matrix computation libraries. In *Proc. 6th International Conference on Parallel Processing and Applied Mathematics (PPAM 2005)*, Sept. 2005. To appear.

[4] H. Kawabata, M. Suzuki, and T. Kitamura. A MATLAB-based code generator for sparse matrix computations. In *APLAS 2004, LNCS 3302*, pages 280–295, Nov. 2004.

[5] K. Kennedy *et al*. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2):387–408, Feb. 2005.

[6] H. Kotakemori, H. Hasegawa, T. Kajiyama, A. Nukada, R. Suda, and A. Nishida. Performance evaluation of parallel sparse matrix–vector products on SGI Altix3700. In *Proc. First International Workshop on OpenMP (IWOMP 2005)*, June 2005. To appear.

[7] LAPACK. http://www.netlib.org/lapack/.

[8] NetSolve. http://icl.cs.utk.edu/netsolve/.

[9] Ninf Project. http://ninf.apgrid.org/.

[10] A. Nishida. SSI: Overview of simulation software infrastructure for large scale scientific applications. In *IPSJ SIG Notes*, 2004–HPC–098, pages 25–30, 2004. In Japanese.

[11] The MathWorks, Inc. http://www.mathworks.com/.

[12] K. Wu and B. Milne. A survey of packages for large linear systems. Technical Report LBNL–45446, Lawrence Berkeley National Laboratory, 2000.